Goal Dependent vs. Goal Independent Analysis of Logic Programs

M. Codish[1] M. García de la Banda[2]
M. Bruynooghe[3] M. Hermenegildo[2]

[1] Dept. of Math. and Comp. Sci., Ben-Gurion Univ., Israel.
codish@bengus.bgu.ac.il
[2] Facultad de Informática, Universidad Politécnica de Madrid, Spain.
{maria,herme}@fi.upm.es
[3] Dept. of Comp. Sci., Katholieke Universiteit Leuven, Belgium.
maurice@cs.kuleuven.ac.be

Abstract. Goal independent analysis of logic programs is commonly discussed in the context of the bottom-up approach. However, while the literature is rich in descriptions of top-down analysers and their application, practical experience with bottom-up analysis is still in a preliminary stage. Moreover, the practical use of existing top-down frameworks for goal independent analysis has not been addressed in a practical system. We illustrate the efficient use of existing goal dependent, top-down frameworks for abstract interpretation in performing goal independent analyses of logic programs much the same as those usually derived from bottom-up frameworks. We present several optimizations for this flavour of top-down analysis. The approach is fully implemented within an existing top-down framework. Several implementation tradeoffs are discussed as well as the influence of domain characteristics. An experimental evaluation including a comparison with a bottom-up analysis for the domain *Prop* is presented. We conclude that the technique can offer advantages with respect to standard goal dependent analyses.

1 Introduction

The framework of abstract interpretation [7] provides the basis for a semantic approach to data-flow analysis. A program analysis is viewed as a non-standard semantics defined over a domain of data descriptions where the syntactic constructs in the program are given corresponding non-standard interpretations. For a given language, different choices of a semantic basis for abstract interpretation may lead to different approaches to analysis of programs in that language. For logic programs we distinguish between two main approaches: "bottom-up analysis" and "top-down analysis". The first is based on a bottom-up semantics such as the classic T_P semantics, the latter on a top-down semantics such as the SLD semantics. In addition, we distinguish between "goal dependent" and "goal independent" analyses. A goal dependent analysis provides information about the possible behaviors of a specified (set of) initial goal(s) and a given logic program. This type of analysis can hence be viewed as mapping a program

and an initial goal description to a description of the corresponding behaviours. In contrast, a goal independent analysis considers only the program itself. In principle the result of such an analysis can be viewed as a mapping from initial goal descriptions to corresponding descriptions of goal behaviours. Consequently a goal independent analysis typically consists of two stages. The first, in which a goal independent mapping is derived from the program; and the second in which this mapping is applied to derive specific information for various different initial goal descriptions.

Traditionally, the standard meaning of a logic program P is given as the set of ground atoms in P's vocabulary which are implied by P. The development of top-down analysis frameworks was originally driven by the need to abstract not only the declarative meaning of programs, but also their behavior. To this end it is straightforward to enrich the operational SLD semantics into a collecting semantics which captures call patterns (i.e. how particular predicates are activated while searching for refutations), and success patterns (i.e. how call patterns are instantiated by the refutation of the involved predicate). Consequently, it is quite natural to apply a top-down approach to derive goal dependent analyses.

Falaschi *et al.* [9] introduce a bottom-up semantics which also captures operational aspects of a program's meaning. This semantics basically consists of a non-ground version of the T_P operator. The meaning of a program is a set of possibly non-ground atoms which can be applied to determine the answers for arbitrary initial goals. This semantics is the basis for a number of frameworks for the bottom-up analysis of logic programs [1, 3]. An analysis based on the abstraction of this semantics is naturally viewed as goal independent.

It is the above described state of affairs which has led to the "folk belief" that top-down analyses of logic programs are goal dependent while bottom-up analyses are goal independent. In fact, bottom-up computations have also been used for query evaluation in the context of deductive databases where "magic sets" and related transformation techniques are applied to make the evaluation process goal dependent. These same techniques have also been applied to enable bottom-up frameworks of abstract interpretation to support goal dependent analysis (see [3] for a list of references). This work breaches the folk belief and suggests that bottom-up frameworks have a wider applicability. In contrast, the practical application of top-down frameworks for goal independent analysis has received little attention. The purpose of this paper is to fill this gap. Moreover, we observe that there are currently a number of fine tuned generic top-down frameworks which are widely available. In contrast, implementation efforts for bottom-up frameworks are still in a preliminary stage. Hence, an immediate benefit of our study is to make goal independent analyses readily available using existing top-down frameworks.

We conclude that the real issue is not top-down vs. bottom-up but rather goal dependent vs. goal independent. As already pointed out by Jacobs and Langen [12], goal dependent analysis can be sped up by using the results of a goal independent analysis, and whether this results in a loss in precision has to do with the characteristics of the abstract domain.

Sections 2, 3 and 4 recall the relevant background and describe some simple transformations enhancing the efficiency of top-down goal independent analysis. Sections 5 and 6 present the main contribution of the paper: an evaluation of the appropriateness of a generic top-down framework (PLAI) for goal independent analysis and the value of a goal independent analysis as a means to speed up a subsequent goal dependent analysis. Sections 7 and 8 discuss the results and conclude.

2 Goal independent analysis in a top-down framework

It is relatively straightforward to apply a top-down framework to provide goal independent analyses much the same as those provided by bottom-up frameworks. To see this consider that, as argued in [9], the non-ground success set obtained by the Falaschi *et al.* semantics is equivalent to the set $\{\ p(\bar{x})\theta \mid p/n \in P \ \textit{ and } \theta \textit{ is an answer substitution for } p(\bar{x})\ \}$. This provides the basis for a naive but straightforward goal independent, top-down analysis. An approximation of the non-ground success set of a program is obtained by performing the top-down analysis for the set of "flat" initial goal descriptions $\langle p(\bar{x}); \kappa_\epsilon \rangle$ where p/n is a predicate in P and κ_ϵ is the (most precise) description of the empty substitution. The same result can be obtained with a single application of the top-down framework by adding to a program P the set of clauses $\{\ \mathit{analyze} \leftarrow p(\bar{x}) \mid p/n \in \mathit{pred}(P)\ \}$ where $\mathit{analyze}/0 \notin \mathit{pred}(P)$. Given the initial call pattern $(\mathit{analyze}; \kappa)$ (with κ any abstract substitution), there is a call pattern $(p(\bar{x}); \kappa_\epsilon)$ for every $p/n \in \mathit{pred}(P)$. We will refer to this transformation as the *naive* transformation and the corresponding analysis as the *naive* analysis.

In this paper we use the top-down framework described in [16] (PLAI). The framework is based on a collecting semantics which captures both success and call patterns. For sharing analyses, the information is represented as lists of lists which appear as comments within the text of the program. The information describes properties of possible substitutions when execution reaches different points in the clause. The information given after the head describes properties of all clause variables after performing head unification. The information given after each subgoal describes properties of all clause variables after executing the clause up to and including the subgoal.

Example 1. Consider the following simple program **P**:

```
mylength(Y,N):- mylength(Y,0,N).
mylength([ ],N,N).
mylength([X|Xs],N1,N):- N2 is N1+1, mylength(Xs,N2,N).
```

The naive transformation adds the following clauses to **P**:

```
analyze:- mylength(X,Y).
analyze:- mylength(X,Y,Z).
```

A goal independent analysis using the *Sharing* domain [11, 15] gives the following:

```
(1)  analyze :-                      %[[X],[Y]]
           mylength(X,Y).            %[[X]]
(2)  analyze :-                      %[[X],[Y],[Z]]
           mylength(X,Y,Z).          %[[X],[Y,Z]]
(3)  mylength(Y,N) :-                %[[Y],[N]]
           mylength(Y,0,N).          %[[Y]]
(4)  mylength([ ],N,N).              %[[N]]
(5)  mylength([X|Xs],N1,N) :-        %[[N1],[N],[X],[X,Xs],[Xs],[N2]]
           N2 is N1+1,               %[[N],[X],[X,Xs],[Xs]]
           mylength(Xs,N2,N).        %[[X],[X,Xs],[Xs]]
```

In the *Sharing* domain [11, 15] an abstract substitution is a set of sets of program variables (represented as a list of lists). Intuitively, each set $\{v_1, \ldots, v_n\}$ specifies that there may be a substitution where the terms bound to the clause variables contain a variable occurring in the terms bound to $v_1, \ldots, v_n$ and occurring in none of the other terms. If a variable v does not occur in any set, then there is no variable that may occur in the terms to which v is bound and thus those terms are definitely ground. If a variable v appears only in a singleton set, then the terms to which it is bound may contain only variables which do not appear in any other term. For example, after executing the recursive call in clause (5) the variables N, N1 and N2 are ground while X and Xs are possibly non-ground. Moreover, if they are non ground, they may possibly share. The analysis provides also the following information indicating the set of call and success patterns:

Atom	Call Pattern	Success Pattern
analyze	[ ]	[ ]
mylength(A,B,C)	[[A],[B],[C]]	[[A],[B,C]]
mylength(A,B)	[[A],[B]]	[[A]]
mylength(A,0,B)	[[A],[B]]	[[A]]
mylength(A,B,C)	[[A],[C]]	[[A]]

Note that while the first three rows give the goal independent information, the other two represent the answers inferred for two specific call patterns which were needed for the abstract computation. □

Observe that the analysis described in Example 1 is inefficient in that it provides information concerning call patterns which are not required in a goal independent analysis. A more efficient analysis is obtained by transforming the program so that all calls in the body of a clause are "flat" and involve only fresh variables. As a consequence, any call encountered in the top-down analysis is in its most general form and corresponds to the call patterns required by a goal independent analysis. This transformation is referred to as the *efficient* transformation and involves replacing each call of the form $q(\bar{t})$ in a clause body by $q(\bar{x}), \bar{x} = \bar{t}$[1] where $\bar{x}$ are fresh variables. The corresponding analysis is called the *efficient* analysis.

[1] Note, however, that in Prolog this transformation can result in a program which

Example 2. Applying the efficient transformation to the program in Example 1 gives:

```
analyze:- mylength(X,Y).        mylength([ ],N,N).
analyze:- mylength(X,Y,Z).      mylength([X|Xs],N1,N) :-
                                       N2 is N1+1,
mylength(Y,N) :-                       mylength(Xsa,N2a,Na),
      mylength(Ya,Ma,Na),               <Xsa,N2a,Na> = <Xs,N2,N>.
      <Y,0,N> = <Ya,Ma,Na>.
```

A goal independent analysis of this program eliminates the last two rows in the table of Example 1. □

As indicated by our experiments (described in the following sections) the "efficient" transformation provides a practical speed-up of up to 2 orders of magnitude (for the domain *Prop*) over the naive approach. As suggested also by Jacobs and Langen [12], we conjecture that the efficient top-down analysis is in fact equivalent to the corresponding bottom-up analysis. In particular, potential loss of precision with respect to a goal dependent analysis is determined by the characteristic properties of the domain.

3 Reusing goal independent information

In this section we illustrate how the results of a goal independent analysis can be (re-)used to derive goal dependent information. For answer substitutions there is no problem as it is well known that the non-ground success set of a program determines the answers for any initial goal. In fact, the same techniques applied in bottom-up analyses can be applied also for top-down goal independent analyses. Moreover, since the call $p(\bar{t})$ is transformed to $p(\bar{x}), \bar{x} = \bar{t}$, the (abstract) unification $\bar{x} = \bar{t}$ with the success pattern for the call $p(\bar{x})$ obtained in the goal independent analysis yields a safe approximation of the success pattern for the original query $p(\bar{t})$. This fact is well known in bottom-up analysis. However our aim is to use the results of the goal independent analysis to derive a safe approximation of all call patterns activated by a given initial call.

Several solutions to this problem are discussed in the literature. These include the magic-set transformation mentioned above as well as the characterization of calls described in [1] and formalized in [10]. Both of these approaches are based on the same recursive specification of calls. Namely: (1) if $a_1, \ldots, a_i, \ldots, a_m$ is an initial goal then $a_i\theta$ is a call if θ is an answer for $a_1, \ldots, a_{i-1}$ (in particular a_1 is a call); and (2) if $h \leftarrow b_1, \ldots, b_i, \ldots, b_n$ is a (renamed) program clause, a is a call, $mgu(a, h) = \theta$ and φ is an answer of $(b_1, \ldots, b_{i-1})\theta$ then $b_i\theta\varphi$ is a call.

---

produces different answers, especially due to the presence of "impure calls" such as $is/2$. Such calls require special care in the goal independent analysis, see discussion at end of section 4.

Our approach is to perform a second pass of top-down analysis to derive the goal dependent information, but using the goal independent information available in order to simplify the process. The idea is to perform the goal dependent analysis in the standard way of the PLAI framework, except for the case of recursive predicates. This framework passes over an and/or graph when analysing a program [2]. In recursive cases, the framework performs several iterations over certain parts of the graph until reaching a fixpoint: when encountering a call $p(\bar{t})$ which is equivalent to an ancestor call, the framework does not analyse the clauses defining $p(\bar{t})$ but instead uses a first approximation (based on results obtained for the ancestor call from the nonrecursive clauses). This initiates an iterative process over a subgraph which terminates when it is verified that a safe approximation is obtained. However, as the results of the goal independent analysis are available, these iterations can be avoided when performing the second pass proposed herein, which can be thus completed in a single traversal of the graph. Note that due to the efficient transformation $p(\bar{t})$ is replaced by $p(\bar{x}), \bar{x} = \bar{t}$. Then, since the success state of the call $p(\bar{x})$ is available from the goal independent analysis, the (abstract) unification $\bar{x} = \bar{t}$ yields a safe approximation of the success state of the call $p(\bar{t})$ and iteration is avoided.

Our approach is similar to that suggested by Jacobs and Langen [11, 12]. The main difference is that they reuse the goal independent information without entering the definition of predicates. In the terminology of [12], the goal independent information is viewed as a "condensed" version of the called predicate and replaces its definition. In contrast, our approach traverses the entire abstract and/or graph constructed in the goal independent phase, even when a more simple 'look-up' could be performed. However, iteration (or fixed point computation) is avoided. Moreover, we obtain information at all program points and for all call patterns encountered in a computation of the initial goal. It is interesting to note that from an implementation point of view, all phases are performed using the same top-down interpreter. We illustrate our approach with an example:

Example 3. Consider a *Sharing* analysis of the following simple Prolog program. The result of the goal independent analysis is indicated next to the program:

```
p([ ],[ ]).
p([X|Xs],[Y|Ys]):- X>Y,q(Xs,Ys).
q([ ],[ ]).
q([X|Xs],[Y|Ys]):- p(Xs,Ys).
```

Atom	Call Pat.	Success Pat.
p(X,Y)	[[X],[Y]]	[[X],[Y]]
q(X,Y)	[[X],[Y]]	[[X],[Y]]

Consider an initial query pattern of the form $\langle p(X, Y); [[Y]] \rangle$ which specifies that X is ground. We illustrate the difference between the standard top-down analysis and the analysis which reuses the results in the above table.

Both analyzers first compute information for the non-recursive clause of $p/2$ obtaining [ ] as the first approximation of the answer. They then consider the second clause obtaining the abstract substitution $[[Y], [Y, Ys], [Ys]]$ (both X and Xs are ground), analyze the built-in $X > Y$ obtaining $[[Ys]]$, and call $q(Xs, Ys)$ with the call pattern $[[Ys]]$. A similar process applies to q with this call pattern: first, the information for the non-recursive clause of q is computed

obtaining [ ] as the first approximation of the answer, then the second clause is considered, obtaining the abstract substitution $[[Y], [Y, Ys], [Ys]]$ (both X and Xs are ground), $p(Xs, Ys)$ is called with call pattern $[[Ys]]$.

At this point, the call pattern is the same as the initial call, hence the *modified* top-down framework analyses $p(A, B), \langle A, B\rangle = \langle Xs, Ys\rangle$ under the abstract substitution $[[A], [B], [Ys]]$. It uses the precomputed table to look up the answer for $p(A, B)$ obtaining $[[A], [B], [Ys]]$ as result of the call. Abstract unification of $\langle A, B\rangle = \langle Xs, Ys\rangle$ gives the abstract substitution $[[Ys, B]]$. Projection on $\{Xs, Ys\}$ gives $[[Ys]]$. The least upper bound of the answers for the two clauses of $q/2$ gives the final result $[[Ys]]$ for $q(Xs, Ys)$. The least upper bound for the two clauses of $p/2$ results in $[[Ys]]$. Note that no fixed point computation is needed.

In contrast, the standard top-down framework takes the current approximation [ ] of the answer for $p(Xs, Ys)$, computes $[[Y]]$ as the approximated answer substitution for the second clause of $q/2$ and takes the least upper bound of this answer and the one obtained for the first clause. This results in $[[Ys]]$ as the approximated answer for the call $q(Xs, Ys)$. The least upper bound for two clauses of $p/2$ gives $[[Ys]]$. Now, a new iteration is started for $p(X, Y)$ since the answer changed during the execution (from [ ] which was the first approximation obtained from the non-recursive clauses, to $[[Y]]$) and there is a recursive subgoal $q(Xs, Ys)$ with call pattern $[Ys]$ which depends on $p(X, Y)$ with call $[[Y]]$; nothing changes during this new iteration and the fixpoint is reached. □

Note that it is still possible that several copies of a same clause are activated, namely when the clause is called with different patterns. The different versions will all be analyzed in the same iteration through the and/or graph whereas the usual top-down framework can iterate several times over (parts of) the and/or graph.

4 Domain dependent issues

There are some domain-dependent issues which can significantly affect the precision of the results obtained. The following example illustrates how, for some domains, a naive top-down analysis can provide a more precise analysis for some programs.

Example 4. Consider a simple (goal independent) type analysis of the following program:

```
rev(Xs, Ys) :- rev(Xs, [ ], Ys).
rev([ ], Ys, Ys).
rev([X|Xs], R, Ys) :- rev(Xs, [X|R], Ys).
```

A reasonable (top-down or bottom-up) goal independent analysis for rev/3 will infer that the first argument is of type 'list' while the second and third arguments are of type 'any'. A naive top-down analysis can infer that both arguments of

rev/2 are of type 'list' because the initial call to rev/3 has a second argument of type list, while a bottom-up analysis, as well as an efficient top-down analysis, will infer that the first argument is of type 'list' and the second of type 'any'. □

The above example illustrates that the precision of an analysis is highly dependent on the ability of the underlying abstract domain to capture information (such as sharing) which enables a good propagation of the property being analyzed.

Jacobs and Langen [12] prove that top-down and bottom-up analyses are guaranteed to be equally precise when they involve an abstract unification function which is *idempotent*, *commutative* and *additive*. Idempotence implies that repeating abstract unification does not change the result. Commutativity allows abstract unification to be performed in any order. Finally, additivity guarantees that precision is not lost when performing least upper bounds. Clearly these conditions impose a restriction on the abstract domain — as a weak domain cannot support an abstract unification algorithm which satisfies these properties. It is interesting to note that while idempotence is satisfied for most of the domains proposed in the literature, the other two properties are not. Consequently, the answer to the question *should we prefer (top-down or bottom-up) goal independent analyses* remains an issue for practical experimentation.

In the remainder of the paper we describe an experimental investigation involving three well known abstract domains, namely, *Prop* [13], *Sharing* [11, 15] and *ASub* [17]. We note that *Prop* satisfies all three of the above mentioned conditions (there is an abstract unification algorithm for *Prop* which satisfies these conditions). For *Sharing*, the first two conditions are satisfied, while *ASub* satisfies only idempotence.

It is interesting to note that additivity in the abstract domain becomes more relevant when performing goal independent analyses. This is because, due to the lack of propagation of information from an initial call, abstract substitutions in an abstract computation tend to contain less information than in the goal-dependent case. Moreover, the accuracy lost when performing least upper bounds becomes more acute as we handle abstract substitutions containing less information. The same holds for commutativity. When more groundness information is available during the abstract computation (due to propagation of information from an initial goal) the inability of the domain to remember dependencies between variables has less effect on accuracy. In fact we observe in [5] that the groundness information obtained with *ASub* is essentially the same as obtained with *Sharing* (for a rich set of benchmarks). We reason that most real Prolog programs tend to propagate groundness in a top-down manner. We expect that (lack of) commutativity will become more relevant in goal-independent analyses although, less important in a naive top-down analyses than in bottom-up or efficient top-down analysis.

Another important issue concerns the behavior of "impure goals". Consider for example an abstract domain which captures definite freeness information. In a standard top-down analysis if we know that the clause
`p(X,Y) :- ground(X), Y=a` is called with X a free variable then we may as-

sume that the clause fails. In contrast, in a top-down goal independent analysis, the initial goal call pattern is always $\{\epsilon\}$ and we must assume downwards closure of all descriptions. Likewise, a goal dependent sharing analysis involving the clause `p(X,Y) :- X==Y` with call pattern $[[X],[Y]]$ may assume that `X==Y` implies that X and Y are ground due to the lack of sharing between X and Y. Such reasoning is not valid in a goal independent analysis.

5 Objectives, experiments and results

Our objective is to illustrate the relative impact of the issues discussed in the previous sections on efficiency and accuracy of goal independent analyses. Our study focuses on a top-down framework, due to its availability. We compare the standard top-down, goal dependent framework with the alternative two phase analysis which first infers goal independent information and then reuses it to obtain goal dependent information for given initial goals.

For goal independent analyses we compare the *naive* and *efficient* approaches described in Section 2. The efficient approach is implemented not as a program transformation but instead by modifying the top-down framework itself which is also modified to keep and reuse goal independent information.

Given the expected dependence of the results on the characteristics of the domains, we have implemented three analyzers, using the domains *ASub*, *Sharing*, and *Prop*. For *Prop* we use the same implementation strategy as described in [4] and provide a comparison with the bottom-up analyses described in [4]. The implementation is based on a technique called "abstract compilation" [8, 18] in which a program is analyzed by applying the *concrete* semantics to an abstraction of the program itself. We note that the bottom-up analysis for *Prop* is based on a highly optimised analyzer which is specific for this type of domain. In contrast, the top-down analysis is performed within the general purpose PLAI framework. Hence, the efficiency results are naturally in favour of the bottom-up analysis. The accuracy results are, as expected, identical.

It should be noted that in our experiments we are using only "strong" domains, i.e. domains that are relatively complex and quite precise. This is done first because they are more likely to represent those used in practice, and also because using goal independent analysis on weak domains is clearly bound to give low precision results.

Tables 1, 2 and 3 respectively present the results of the experiments performed with the *Prop*, *Sharing* and *Asub* domains. The benchmark programs are the same as those used in [4] and in [6] for evaluating the efficiency of their bottom-up approach. All analyses are obtained using SICStus 2.1 (native code) on a SPARC10. All times are in seconds. The columns in the respective tables describe the following information:

Name: the benchmark program and the arguments of the top-level predicate.
GI: the results for the goal independent analyses

- **BU**: time for the bottom-up analyzer described in [4].
  Available only for Prop – Table 1.

- **GI**^{ef}: time for the efficient top-down goal independent analysis.
- **GI**^{n}: time for the naive top-down goal independent analysis.
- **Size**^{n}: A measure of the average and maximal (between parenthesis) sizes of the results given by the naive top-down goal independent analyses.
  For *Prop* (Table 1), the number of disjuncts in the resulting disjunctive normal forms and for *Sharing* and *ASub* (Tables 2 and 3), the number of variables in the resulting abstract substitutions.
- Δ: the percentage of *predicates* for which the analysis using **GI**^{ef} is less accurate than that obtained by **GI**^{n}.
  Only in Tables 2 and 3 (for Prop both techniques give identical results).

GD^{reuse}: the results for the goal dependent analyses which reuse the (efficient) goal independent information:

- **Query**: some example call patterns (for *Prop*, a propositional formula on the variables of the top-level predicate).
- **Tm**: the time.
- **RP**: number of look-ups (in the results of the goal independent phase)
- **Size**: The same measure of the size as above, but this time it only takes into account the answers obtained in the goal independent phase which have been looked-up. This information is included to give a rough idea of the complexity of the abstract unification operations involved.

GD^{standard}: results for the standard top-down, goal dependent analyses in computing the goal dependent information for the indicated query.

- **Tm**: the time.
- > 1: number of fixed point computations that take more than one iteration. These are the non-trivial computations.
- > 2: number of fixed point computations which take more than two iterations. Note that the last iteration usually takes much less time than the others. So these computations are bound to be more costly than those which involve only two iterations.

Δ: the % of program *points* at which the information inferred by the **GD**^{reuse} is less accurate than that obtained by the standard **GD**^{standard} approach.
Only in Tables 2 and 3 (for Prop both techniques give identical results).

6 Discussion of the results

We first compare the two approaches proposed for gathering goal independent information using a top-down framework. The results for *Prop* and *Asub* show that GI^{ef} is consistently considerably better than GI^{n}. This is because the abstract unification functions for those domains are relatively simple, and thus the overhead due to the additional operations introduced by the *efficient* transformation is always smaller than the call computation overhead in GI^{n}. On the other hand, in the results for *Sharing* although GI^{ef} is considerably faster in

	GI				GD^{reuse}				GD^{standard}		
Name	BU	GI^{ef}	GI^{n}	Size	Query	Tm	RP	Size	Tm	> 1	> 2
reverse	0.01	0.04	0.28	2.0 (2)	A	0.04	3	2.0 (2)	0.04	0	0
(A,B)					*true*	0.26	17	2.0 (2)	0.49	10	5
qsort	0.01	0.04	0.39	1.7 (2)	A	0.33	19	1.4 (2)	0.33	0	0
(A,B)					*true*	0.74	31	1.3 (2)	0.74	0	0
queens	0.03	0.11	0.70	2.0 (3)	A	0.18	10	2.5 (3)	0.18	0	0
(A,B)					*true*	0.78	40	2.6 (3)	1.07	6	2
pg	0.04	0.39	2.47	1.3 (2)	A	0.41	17	1.5 (2)	0.42	0	0
(A,B)					*true*	0.41	17	1.5 (2)	0.42	0	0
plan	0.04	0.21	1.40	1.9 (5)	A	0.47	9	2.5 (4)	0.47	0	0
(A,B)					*true*	0.46	9	2.5 (4)	0.47	0	0
gabriel	0.08	0.45	4.38	1.9 (4)	A	3.05	162	3.1 (4)	6.04	57	17
(A,B)					*true*	3.06	162	3.1 (4)	6.05	57	17
cs	0.40	2.49	16.09	2.2 (6)	A	1.25	31	2.9 (4)	1.37	3	1
(A)					*true*	1.72	32	2.9 (4)	1.82	3	1
press	0.40	3.24	30.50	2.3 (8)	A	20.62	966	2.6 (4)	35.85	124	35
(A,B)					*true*	20.65	966	2.6 (4)	37.85	115	32
read	0.32	2.71	33.27	1.9 (9)	A	20.15	355	2.2 (9)	56.96	214	100
(A,B)					*true*	20.17	355	2.2 (9)	57.05	214	100
peep	0.47	4.10	37.07	2.6 (10)	A	7.65	82	2.2 (4)	9.94	37	7
(A,B,C)					*true*	29.66	370	2.7 (4)	70.99	181	81

Table 1. *Prop* results

most cases, there are others where this difference is not as large, and a few in which GI^{ef} in fact performs slightly worse than GI^{n}. This is explained by the complexity of the abstract unification function for *Sharing*.

From the precision point of view, of course, for *Prop* there is no loss of precision. Relatively high precision is maintained in *Sharing*, while some more important loss appears for *Asub*. This reflects the fact that *Asub* is a weaker domain than *Sharing* w.r.t. the three basic properties. Thus, GI^{ef} appears to present a good precision / cost compromise.

We now compare GD^{reuse} (the goal dependent phase with goal independent information available), with a standard goal dependent computation. GD^{reuse} is almost consistently faster (or equal) to GD^{standard}, and the difference in speed is proportional to the number of fixed points avoided with respect to GD^{standard} and the complexity of these, as can be observed from the "> 1" and "> 2" columns. This last column seems to be the one that best predicts the differences in performance: any time this number is high the differences are significant. This result would be expected since this column indicates the number of "heavy" fixed point computations in the GD^{standard} approach.

The exception is in the *Asub* analysis. There, for some programs, the analysis is less precise in more than 50% of the program points. A consequence of this is

Name	GI				GD^{reuse}				GD^{standard}			Δ
	GI^{ef}	GI^{n}	Size^{n}	Δ	Query	Tm	RP	Size	Tm	>1	>2	%
init_susbt (X,Y,Z,W)	0.9	173.5	3.1 (5)	0	[[Z],[W]]	0.2	9	4.2 (5)	0.2	0	0	0
					[[Y],[Z],[W]]	0.7	15	4.1 (5)	0.9	6	1	0
					[[X],[Y],[Z],[W]]	98.1	21	4.7 (5)	193.7	21	4	0
serialize (X,Y)	0.7	3.0	2.3 (4)	0	[[Y]]	2.8	14	3.4 (4)	3.0	8	0	0
					[[X],[Y]]	2.9	14	3.4 (4)	3.1	8	0	0
					[[X],[X,Y],[Y]]	2.9	14	3.4 (4)	3.1	8	0	0
map-color (X,Y,Z,W)	1.4	1.9	2.1 (3)	0	[[Y],[Z],[W]]	1.5	5	2.6 (3)	3.1	8	0	0
grammar (X,Y)	0.1	0.1	1.9 (3)	0	[[X],[Y]]	0.1	0	0 (0)	0.1	0	0	0
					[[X],[X,Y],[Y]]	0.1	0	0 (0)	0.1	0	0	0
browse (X,Y)	3.9	14.0	2.3 (5)	0	[ ]	13.6	18	2.5 (5)	16.4	9	0	0
					[[X],[Y]]	0.2	10	2.1 (3)	0.4	8	0	0
					[[X],[X,Y],[Y]]	0.2	9	2.1 (3)	0.3	6	0	0
bid (X,Y,Z)	0.5	1.4	1.5 (3)	0	[ ]	0.3	7	2.9 (3)	0.3	0	0	0
deriv (X,Y,Z)	0.8	1.9	2.5 (3)	0	[[Z]]	0.9	35	2.7 (3)	0.9	0	0	0
					[[Y],[Z]]	0.9	35	2.7 (3)	0.9	0	0	0
rdtok (X,Y)	0.7	1.5	2.0 (5)	0	[[X],[Y]]	1.2	47	2.0 (3)	2.0	25	13	0
					[[X],[X,Y],[Y]]	1.2	47	2.0 (3)	2.0	25	13	0
read (X,Y)	10.6	206.0	2.4 (11)	4	[[Y]]	1.5	22	4.5 (6)	1.5	18	11	0
					[[X],[Y]]	66.4	73	4.6 (6)	257.9	270	115	0
boyer (X)	3.7	7.5	2.3 (5)	0	[ ]	1.7	15	2.5 (3)	4.0	45	19	0
					[[X]]	1.7	13	2.6 (3)	4.0	44	18	0
peephole (X,Y)	33.4	19.4	3.3 (6)	0	[[Y]]	4.1	60	2.1 (3)	7.3	28	7	0
					[[X],[Y]]	11.1	63	2.1 (3)	19.8	36	10	0
ann (X,Y)	418.1	381.8	3.3 (12)	6	[[X],[Y]]	22.2	69	2.9 (6)	27.8	40	11	2.4
					[[X],[X,Y],[Y]]	22.1	69	2.9 (6)	27.7	39	10	2.4

Table 2. *Sharing* results

that domain elements are a lot larger (imprecision increases the number of pairs) and that their processing is more time consuming. In some cases the difference is substantial enough to undo the effect of saved fixed point iterations.

On the other hand, while GD^{reuse} is almost consistently faster than GD^{standard}, the difference is not as big as one might expect. This is due to the fact that a very efficient fixed point is being used in GD^{standard}, which, by keeping track of data dependencies and incorporating several other optimizations, performs very few fixed point iterations – often none.

From the point of view of the precision of the information obtained the results are identical for *Prop*, and slightly different for the slightly weaker *Sharing* domain. This precision is quite surprising and implies that not much information is lost in least upper bound operations, despite the weakness of *Sharing* in

	GI				GD^{reuse}				$GD^{standard}$			Δ
Name	GI^{ef}	GI^{n}	$Size^{n}$	%	Query	Tm	RP	Size	Tm	> 1	> 2	%
init_susbt (X,Y,Z,W)	0.2	0.4	3.14 (5)	0	([X,Y],[])	0.3	9	4.2 (5)	0.4	5	0	0
					([X],[])	0.3	12	3.8 (5)	0.5	8	0	0
					([],[])	0.3	9	4.4 (5)	0.4	6	0	0
serialize (X,Y)	0.2	0.2	2.3 (4)	0	([X],[])	0.1	8	2.8 (4)	0.2	5	1	12.5
					([],[])	0.1	8	2.8 (4)	0.2	5	1	12.5
					([],[[X,Y]])	0.4	9	2.8 (4)	0.4	6	1	12.5
map-color (X,Y,Z,W)	0.2	0.3	2.6 (4)	0	([X],[])	0.3	6	2.5 (3)	0.3	2	0	0
grammar (X,Y)	0.0	0.0		0	([],[])	0.0	0	0 (0)	0.0	0	0	0
					([],[[X,Y]])	0.1	0	0 (0)	0.1	0	0	0
browse (X,Y)	0.3	1.1	2.0 (4)	11.8	([],[])	0.6	18	2.4 (4)	0.5	7	0	71.4
					([],[])	0.1	9	1.6 (3)	0.2	6	0	0
					([],[[X,Y]])	0.6	13	1.5 (3)	0.7	9	0	0
bid (X,Y,Z)	0.3	1.0	1.8 (4)	5	([],[])	0.5	8	2.6 (3)	0.3	0	0	76.2
deriv (X,Y,Z)	0.6	2.1	2.5 (3)	0	([X,Y],[])	3.1	72	2.3 (3)	0.8	0	0	91.1
					([X],[])	3.1	72	2.2 (3)	0.8	0	0	91.1
rdtok (X,Y)	0.7	1.0	2.6 (4)	33.3	([],[])	1.0	43	2.9 (4)	1.4	23	12	17.9
					([],[[X,Y]])	1.0	43	2.9 (4)	1.4	23	12	17.9
read (X,Y)	2.1	9.3	2.5 (10)	4	([X],[])	4.8	61	3.2 (4)	1.8	18	11	74.5
					([],[])	3.7	46	3.3 (4)	10.4	121	50	0
boyer (X)	0.7	1.1	2.3 (5)	0	([X],[])	0.8	15	2.1 (3)	1.4	45	19	0
					([],[])	0.8	15	2.2 (3)	1.4	45	19	0
peephole (X,Y)	1.8	2.9	3.6 (6)	0	([X],[])	1.7	58	2.2 (4)	2.5	21	3	0
					([],[])	1.9	58	2.2 (4)	3.0	25	6	0
ann (X,Y)	2.9	11.5	3.0 (10)	3	([],[])	3.9	79	2.8 (6)	5.1	37	9	6.5
					([],[[X,Y]])	5.4	94	2.8 (6)	6.6	40	10	6.5

Table 3. *Asub* results

performing them. This seems to imply that the information being "LUBed" is highly consistent (i.e. all clauses give similar information - while one can easily write artificial predicates not having this property, it is not unexpected that such predicates are rare in real programs). The case of the *read* benchmark in the *Sharing* analyzer would appear surprising in the sense that although some information is lost by GI^{ef} there is no loss of information after the GD^{reuse} pass w.r.t. $GD^{standard}$. This is due to the fact that the predicate that changes is not used in the goal dependent computation for the query patterns analyzed. Less surprising is the fact that the weaker *Asub* domain presents more differences in the information obtained.

In order to perform a completely fair comparison of the goal independent and goal dependent approaches one should really compare the $GD^{standard}$ time with the sum of the GI^{ef} (or GI^{n}) time and the GD^{reuse} time, since to obtain

information with GD^{reuse} it is necessary to perform the GI^{ef} analysis first. In this case the results are mixed in the sense that there is still a net gain in performance for benchmarks and call patterns which require several complex fixed point iterations from $GD^{standard}$, but there is also a net loss in other cases. This is surprising and shows again that $GD^{standard}$ is quite good at avoiding fixed point iterations.

The GI^{ef}+GD^{reuse} approach is interesting in that it arguably provides more predictable execution times (although still highly dependent on the query pattern), sometimes avoiding cases in which $GD^{standard}$ incurs larger overheads due to complex fixed point calculations. The combined GI^{ef}+GD^{reuse} analysis seems to be of advantage in the special case of programs that reuse their predicates in many ways and with different call patterns. However, our results show that this is not often the case, at least for our benchmarks. Thus, $GD^{standard}$ seems to end up probably winning when analyzing normal isolated programs. A further advantage for $GD^{standard}$ is that it is quite general in that it does not require any special strengths from the domains to keep the precision that one would expect from them.

The overall conclusion seems to be that the combined GI^{ef}+GD^{reuse} analysis is specially suited for situations where the results obtained in the goal independent phase have the potential of being reused many times. A typical example of this is library modules. They may be preanalyzed to obtain goal independent information which is stored with the module. Then only the GD^{reuse} pass is needed to specialize that information for the particular goal pattern corresponding to the use of the library performed by the program that calls it.

7 Conclusions

Our experiments with the *Prop* domain indicate that the efficient version of our goal independent analysis making use of the generic top-down framework is a viable alternative to a goal independent analysis using a bottom-up implementation as its speed is within a factor of 10 of a highly tuned ad hoc implementation of a bottom-up analysis for prop.

A goal dependent analysis which can use the results of a goal independent analysis is, for programs where the difference in precision is insignificant, consistently faster than a goal dependent analysis which starts from scratch. Also the analysis time is more closely related to the program size and becomes more predictable. This is due to the fact that the goal dependent analysis starting from scratch can require an unpredictable amount of iterations before reaching a fixpoint for its recursive predicates. While the precision is the same when abstract unification is idempotent, commutative and additive, the loss of precision is quite small for the *Sharing* domain which is not additive. The reason seems to be that the different clauses of real program predicates usually return very similar abstract states, such that the lub operator in practice rarely introduces a loss of precision. On the other hand, the loss of precision can be substantial in the *Asub* domain which also violates the commutativity condition.

Finally, the *Sharing* domain illustrates a case where, for some programs, the goal independent analysis can take an unexpected long time. This is caused by the peculiarities of the sharing domain. The size of an abstract state is in the worst case exponential in the number of program variables, this worst case typically shows up in absence of (groundness) information, and is much more likely to occur in a goal independent analysis than in a goal dependent analysis. Indeed, in the latter case, the information coming from the typical queries curtails the size of the abstract states.

References

1. R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15:133–181, 1993.
2. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
3. M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Journal of Theoretical Computer Science*, 124:93–125, 1994.
4. M. Codish, B. Demoen. Analysing Logic Programs using "`Prop`"-ositional Logic Programs and a Magic Wand. In *Proceedings International Logic Programming Symposium*. Vancouver, October 1993. MIT Press.
5. M. Codish, A. Mulkers, M. Bruynooghe, M. de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*. ACM, 1993.
6. M. Corsini, K. Musumbu,A. Rauzy,B. Le Charlier. Efficient bottom-up abstract interpretation of Prolog by means of constraint solving over symbolic finite domains. *Proc. of the Fifth International Symposium on Programming Language Implementation and Logic Programming*. Tallinn, August 1993, LNCS 714, Springer Verlag.
7. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Rec. 4th Acm Symp. on Prin. of Programming Languages*, pages 238–252, 1977.
8. S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.
9. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modelling of the Operational Behaviour of Logic Programs. *Theoretical Computer Science*, 69:289–318, 1989.

10. J. Gallagher, M. Codish, E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6 (1988) 159-186.
11. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.
12. D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.
13. K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. *Information Processing*, pages 601–606, April 1989.
14. K. Marriott, H. Sondergaard, and P. Dart. A Characterization of Non-Floundering Logic Programs. In *Proc. of the 1990 North American Conference on Logic Programming*. MIT Press, 1990.
15. K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.
16. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.
17. H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.
18. R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699, Seattle,Washington, August 1988. MIT Press.